

APPENDIX A

<!--

Pixxa Exchange Protocol XML DTD

Copyright (c) 1998-2000 Farshad Nayeri. All Rights Reserved.

-->

<!-- ===== introduction -->

<!--

This document specifies the Pixxa Exchange Protocol (PXP), a communication protocol for synchronizing a collection of items in two independent agents. Pixxa Exchange Protocol builds on top of standard transport protocols (TCP/IP, HTTP) and encodings (XML, GIF, JPEG, URL, and MIME standards.)

The Pixxa system consists of:

users
media items

Each user has:

id (or a username)
collection of items

Each media item has:

id: used for identity comparison

name: short name of the object

content: where the content of this media item resides,
e.g., the src of the IMG tag

contenttype: what is the mime type for this item. For now,
assume it is image/jpeg image/gif

title: the title screen displayed for this item

page: the source page where we got this item

details: the fine print on this item, e.g., copyright info on
images

The goal of the Pixxa Exchange Protocol is to have the client and
the server share the same "knowledge" as to

the items in the collection for a particular user. The client and server should be able to operate with a partial collection at any time. Note that the media in a collection may not reside on the Pixxa server; they may be anywhere on the Internet.

A media item may be 'materialized' which means that its content has

been copied to the client-side cache. The client-side cache is persistent across restarts of the client. Note that the same user may have a client on multiple machines; these will effectively be replicated but they may have different media items materialized.

A sound sameness criteria for media items will be difficult to define formally, especially across different formats. For now, we

assume that each media item has a unique id. Ultimately, we would

like collections to be true sets where only one instance of the same

media item exists. Somewhere along the two ends of the spectrum lies

the approach of using some form of fingerprints for media item equality. (Obviously we don't want to compare the entire bits of media items.)

Each media item has a 'preference rating' which describes how well

the user likes that media item:

| | |
|----------|---------------------------------------|
| 0 | => ambivalent or unrated (don't care) |
| positive | => like |
| negative | => dislike |

Each media item starts with zero rating. Items may be promoted (or

demoted) by the user which increases (or decreases) their preference

rating by one unit. Items with negative rating do not get displayed

on the client unless explicitly requested. The higher the rating of the media item the more frequently it is displayed.

This rating information is implicitly communicated as normal part of PXP's operation.

-->

```
<!-- ===== conventions -->
<!--
```

Section tags, such as "rendezvous", use long names whereas item tags, such as item-ref, use short names.

Tags usually end in:

-ref a reference to something; this is a form of declaration to let the other side know that this object lives on this side.

-def a definition of something, usually result of a -req from the other side. Sometimes client or server voluntarily define something, e.g., username and password.

-req a request for something, the other side should send it next time

-->

```
<!-- ===== protocol basics -->
<!--
```

Pixxa client and server communicate via HTTP POST requests and HTTP responses carrying XML documents conforming to the

PXP XML DTD.

A typical interaction between the client and server is as follows:

Client	Server
0.	>>>> empty rendezvous >>>>>
1.	rendezvous info, <<<<< email,passwd req. <<<< generic matches
2.	>>>> rendezvous info, email, passwd def like/dislikes >>>>>
3.	rendezvous info, <<<<< latest matches, <<<< schema changes
4.	>>>> rendezvous info, like/dislikes >>>>>
5.	rendezvous info, <<<<< latest matches, <<<< schema changes
 repeat 4 and 5

Explanation:

0. A fresh client sends empty rendezvous to the server when it gets started.
1. The server requests authorization information (email, password) and sends back some generic matches (since it doesn't yet know who the client is.)
2. The client will pass back userid and password, and maybe some like dislikes.

3. The server will send back a set of changes for the latest matches

to the client in response to this request. In case the system has

had any schema changes (i.e., media items which have been deleted or

modified) the changes are also communicated. Finally, through the

rendezvous info, the server also tells the client when to

contact it again and what the client needs to present to the server.

4. The client sends the latest likes and dislike sets, including

the rendezvous info it got from the server.

5. Repeat steps 4 and 5.

-->

<!-- ===== pxp: exchange unit -->

<!--

A PXP transmission is a kind of rendezvous by two agents; the goal of the agents is to synchronize their information about some external resources (such as media instances on the internet.)

During the rendezvous, the each side exchanges information about its state and requests information to be sent in the next rendezvous. PXP is designed to allow agents to progress independently with coarse communication that are few and far in between.

A complete rendezvous is a result of two PXP messages, a request by

a client is satisfied with a response from the server. Both client

and server use PXP to exchange the information, each carrying information about the changes in the collection. Each rendezvous is tied to the next one because the server issues a

rendezvous ticket which can be used for a certain period of time

-->

```
<!ELEMENT pxp (rendezvous?, variables?, special?, instances?, reports?)>
```

```
<!ATTLIST pxp version      NMOKEN      #IMPLIED
           role          (client|server|provider|archive) "provider">
```

<!--

A pxp message may include:

| | |
|------------|---|
| version | version information, currently 3.1 |
| role | whether the message is sent by an agent taking on a client or server role. |
| rendezvous | information on the last rendezvous |
| variables | variable binding requests and responses |
| special | meta-information about client's collection |
| instances | requests for actions that should be performed by the other side on instances, e.g., insertion and deletion |
| reports | briefs the other side about what happened during various actions, for example, whether certain media items could not be accessed. |

A pxp message can carry information that have different but similar roles. Eventually there may be multiple, related definitions for these roles but for simplicity we will embed them in the same

definition for now.

server an active server which manages pxp information from various places. This mode is used for server communicating back to the client.

```
pxp role="server"
rendezvous
rz-def
variables
var-req
special
var-def
instances
inserts
deletes
updates
defines
```

client client that merely views and marks items. This mode is used for client communicating to the server. The following tags are legal in client role:

```
pxp role="client"
rendezvous
rz-ref
variables
var-def
reports
rppt-def
```

provider a content provider, for example, a site that has some gifs and wants to create a collection from them without involving the server extensively.

```
pxp role="provider"
special
var-def
instances
defines
```

archive an archive file, for example, saved by the client in between client sessions. For uniformity, archive files use a dialect of the protocol to ease interoperability.

```
pxp role="archive"
rendezvous
  rz-def
  variables
    var-def
    var-ref
  special
    var-def
  reports
    rppt-def
  instances
    inserts
    deletes
    updates
    defines

-->

<!-- ===== ren
dezvous -->

<!ELEMENT rendezvous ( rz-def | rz-ref )? >

<!ELEMENT rz-def EMPTY >
<!ATTLIST rz-def host          NMTOKEN #IMPLIED
          time          CDATA    #REQUIRED
          delaymin     NMTOKEN #IMPLIED
          delaymax     NMTOKEN #IMPLIED
          ticket        NMTOKEN #REQUIRED>

<!ELEMENT rz-ref EMPTY >
<!ATTLIST rz-ref ticket NMTOKEN #REQUIRED
          time    CDATA    #IMPLIED
          info    CDATA    #REQUIRED>

<!--
```

The rendezvous statement specifies the timing of the communication between client and the server. Obviously, clients can

access servers at will, as they do in HTTP. However, this rendezvous mechanism allows the server to manage its resources (bandwidth, processor time, and memory) by adjusting how often

a

client makes accesses to the server.

A rendezvous record either is either defined by server in order to

communicate the next time the client should try approaching the server (rz-def), or defined by a client to specify when the last rendezvous was (rz-ref). Rendezvous info includes:

| | |
|----------|---|
| host | where to go for the next rendezvous |
| time | the current server time using HTTP format
e.g., 14 January 2000 12:22:33 EST |
| delaymin | the minimum time to wait before contacting the server |
| delaymax | the maximum time to wait before contacting the server |
| ticket | present this at the next rendezvous |
| info | miscellaneous variable bindings sent by the client
including "uptime= ", where uptime is the time in seconds since the client started up |

t

A "fresh" client may pass an empty rendezvous statement to the server (i.e., <rendezvous></rendezvous> to denote that it doesn't have any previous rendezvous information.)

-->

```
<!-- ===== var
```

iables -->

```
<!ELEMENT variables ( (var-def | var-req) * ) >
```

```
<!--
```

Variables statements request variable bindings, passing the required information for a dialog (var-req, usually done by the server) or for the binding for a variable to come back (var-def .)

Each response from the server may carry one or more variable requests, which turn to dialog displays for a client. Each dialog

is marked with the rendezvous information passed down when the server initially requested the dialog. The client will prompt the

user with this dialog. If the user responds in the specified period

of time, the user's response is sent to the server in the next rendezvous.

If the user doesn't respond to a dialog, the corresponding dialog

response is not sent to the server. If this dialog response is crucial

for server operation (for example, a confirmation password of a newly

registered user), it may respond back again for the same prompt. This

process is continued until the requested information is supplied.

```
-->
```

```
<!ELEMENT var-def EMPTY >
```

```
<!ATTLIST var-def var    CDATA    #REQUIRED
                           val    CDATA    #REQUIRED>
```

```
<!ELEMENT var-req EMPTY>
```

```
<!ATTLIST var-req var          CDATA          #REQUIRED
                           default      CDATA          #IMPLIED
                           prompt       CDATA          #IMPLIED
```

```

details      CDATA      #IMPLIED
delay       NMOKEN      #IMPLIED
type        (text|password|inform|confirm)
            'text' >

```

<!--

A var-def binds the value of a variable. Its attributes are:

```

var      name of the variable
val      the value for a variable

```

Var-defs from the client are usually the result of a previous var-req by the server. However, this may not always be the case

the protocol allows for variables to be bound voluntarily by the client (for example, to pass runtime platform info.)

A var-req requests a new variable to be assigned:

```
var      name of the variable
```

```
default the default value for the variable
```

```
prompt  a short (one or two word) prompt, e.g., Username
```

```
details the fine print for the question
```

```
delay   how long should the question be displayed
```

```
type    hint for the client as to how it should gather the requested variable. Note that the ultimate choice
```

of the dialog is up to the client. The following are valid types

:

```
text      allow the user to type in answer
```

```
password  ask the question, allowing user to type in answer
```

o typein in "blind" mode; the response should be encrypted.

inform
for the specified
r interaction.

just display the detail information
period of time without requiring use
No variable binding is expected.

confirm
ed
el.
"cancel".

display the question for the specific
period of time, expecting ok or cancel.
The result should be either "ok" or

choose
he

display a list of options, and let the
user choose one. Treat default value
list of choices.

as a comma-separated
select
he
em.

display a list of options, and let the
user choose some, all, or none of them.
Treat default as a comma-separated
list of choices, and return a comma-
list of the selected items.

separated

-->

<!-- ===== -->

<!ELEMENT special (var-def*) >

<!--

A special element contains zero or more variable definitions.
The server
sends a special element to provide the client with meta-information
about
the collection. Variables bound within a special element might
include:

screenplays list of screenplay mnemonics, in descending order of preference
 For example, screenplays="slideshow thumbnails" means that the client should use the slideshow screenplays in the collection, if that screenplay is available. If not, it should try to use the thumbnails screenplay, and so on.

params parameters passed to the screenplay, a set of name=val bindings.

size the size of the collection
 when not specified, the collection is unbounded

origin sequencing origin; where to start in the collection.
 this option may be used by the server to transport the sequence from one workstation to another

idleratio specifies how aggressively client should download the collection

After completing a download, client pauses before beginning the next download. The length of the pause is computed as

`pause = idleratio * last_download_duration`

where `last_download_duration` is the time needed to complete the most recent successful download. `idleratio` is a non-negative number; the

smaller it is, the more aggressively the client will attempt to download the collection.

increment the increment for sequencing

Items are indexed starting with zero. The client may sequence through the collection using the following formula:

```

i[ 0 ] = origin
i[ n ] = ( i[n-1] + increment ) MOD size
i[n-1] = ( i[n] - increment ) ... if
i[n] >= increment ...
i[n-1] = ( i[n] - increment ) + size ... if
i[n] < increment ...

```

If $i[n]$ is not materialized, it is skipped; the client repeats this until an item has materialized.

For example,

```

{origin=0,increment=1}          => sequential
scan
(origin=0,increment=largeprime) => random scan
of entire set
-->

```

```

<!-- ===== reports -->

```

```

<!ELEMENT reports (rprt-def) * >

```

```

<!--

```

Reports are the primary method for a client to communicate with the server. The syntax for reports has been unified so that it can easily be extended for new uses.

```

-->

```

```
<!ELEMENT rppt-def (item-ref*) >
<!ATTLIST rppt-def type          NMTOKEN #REQUIRED
               options        CDATA  #IMPLIED>
```

```
<!--
```

Items describe resources on the web. Each item has one or more facets, e.g., an associated thumbnail or an associated image. The idea is that we can extend the kinds of facets, e.g., to support sound files, quicktime movies, and so on, by adding new facets.

A report definition may have a:

| | |
|---------|--|
| type | what type of report, see below for a list |
| options | specific options for this instance of the report |

A report may have one or more item definitions or references.

```
-->
```

```
<!ELEMENT item-def (facet*)>
<!ATTLIST item-def id          CDATA #REQUIRED
               pos         NMTOKEN #IMPLIED

               title       CDATA #IMPLIED
               details    CDATA #IMPLIED
               page       CDATA #IMPLIED
               rating     CDATA "0"
               info       CDATA #IMPLIED

               fgcolor    CDATA #IMPLIED
               bgcolor    CDATA #IMPLIED
               hicolor    CDATA #IMPLIED
               uncolor    CDATA #IMPLIED

               relmod     NMTOKEN #IMPLIED >

<!ELEMENT item-ref EMPTY>
<!ATTLIST item-ref id        CDATA #REQUIRED
```

```

        note          CDATA  #IMPLIED
        relmod        NMTOKEN #IMPLIED >

<!ELEMENT facet    EMPTY>
<!ATTLIST facet   kind      CDATA  #REQUIRED
                   src       CDATA  #REQUIRED
                   info     CDATA  #IMPLIED
                   mimetype CDATA  #IMPLIED >

```

<!--

Attributes for items:

| | |
|-------------------|--|
| id | unique identifier for this item |
| pos | the position of the item within the collection |
| | default is one larger than the index of the last materialized picture. |
| title | the name of this item |
| detail | the fine details for this item
default is "Find out more about
<a href=http://[serverhost]
/pixxa/client/action/detail
>[item-title]." |
| page | the page to follow for this item
default is http:<a href=http://[serverhost]
] |
| find?id=[item-id] | /pixxa/client/action/page- |
| rating | the rating for this item;
default is zero |
| note | in item-ref marks the item with a specific |
| note,
is | for example, what type of failure caused this |

item to be in a problem report.

| | |
|---------|------------------------------------|
| fgcolor | foreground color (format: #rrggbb) |
| bgcolor | background color (format: #rrggbb) |
| hicolor | highlight color (format: #rrggbb) |
| uncolor | disabled color (format: #rrggbb) |

| | |
|--------------------|--|
| relmod | the number of seconds between the latest r |
| endezvous and when | this item was last changed. Suppose the cl |
| ient | makes a change to the rating of an |

item. Sometime later the client receives a notification that the rating should change again, reverting the rating back to normal. (This may have been caused by the user's use of another client, or just beca

the server has stale information on this i
for whatever reason.)

In these cases, the client can find out approximately when the item was changed in client-local time (using relmod and the client-local time of the latest rendezvous

and then keep the rating change that happe
later.

An item may contain zero or more facets. A facet describes a different presentation of the item. Each facet contains:

| | |
|------|---|
| kind | what type of facet, legal values include:
- thumb
- image
- logo
- flash
- sound |
|------|---|

| | |
|-----|--|
| src | the source url for the content of this fac |
| et. | |

| | |
|----------|---|
| info | kind-specific info about the facet (reserved for future use) |
| ed | |
| mimetype | mime type for the content. If none is specified |
| ified | it is up to the client to decide. |
| alt | alternative text for the facet. If no alt is specified, the item-def's title must be used |
| used | as a default. |

Here is an example of an item-def:

```

<item-def id="amazon_com"
          title="Amazon.com"
          details="Amazon.com: Earth's biggest bookstore."
          page="http://www.amazon.com" >

    <facet kind="thumb"
          src="http://www.amazon.com/g/associates/logos2000/1
26X32-b-logo.gif"
          mimetype="image/gif" />

    <facet kind="image"
          alt="Amazon.com Logo Image"
          src="http://www.amazon.com/g/associates/logos2000/1
49X45-b-logo.gif"
          mimetype="image/gif" />

</item-def>

```

In the case of the thumb facet, its alt uses the default, which is the title from the enclosing item-def.

-->

<!-- ===== rep

```
ort types -->
```

```
<!--
```

The following is a list of valid types for reports:

```
rating
duplicate_item_insert
unknown_item_update
unknown_item_delete
update_conflict
stale_item
stale_everything
unknown_item_referenced
unknown_variable_referenced
refreshed_item
```

```
-->
```

```
<!-- ===== rating
reports -->
```

```
<!--
```

A rating report indicates that the users' likes and dislikes. The options set to "-1", "+1" or "0" affect all items referenced in the report.

```
<rprt-def type="rating" options="-1">
  <item-ref id=...>
</rprt-def>
```

```
-->
```

```
<!-- ===== management
reports -->
```

```
<!--
```

Reports are sent by a client which has trouble performing certain item operations, for example, updating items.

```
<rprt-def type="unknown_item_deleted">
  <item-ref id=...>
</rprt-def>
```

See the list of report types and different actions to find out more about problem reports.

-->

```
<!-- ===== media failure reports -->
```

```
<!--
```

When the client can't reach a media item, it marks the item to be reported in a "media failure" report in the next rendezvous.

```
<rprt-def type="media_failure">
  <item id=... note="404 NotFound">
</rprt-def>
```

The note for the item carries the HTTP causing the media failure when possible.

-->

```
<!-- ===== stale item reports -->
```

```
<!--
```

Stale item reports are sent as part of client requests; the server usually refreshes the entire value for the item. This is an unusual request by the client; there is evidently something wrong with the data gathered by the client.

```
<rprt-def options="stale_item">
```

```
<item-ref id=...>  
</rprt-def>
```

-->

```
<!-- ===== stale everything reports -->
```

<!--

The entire client cache is stale, invalid, or empty. Client should receive the entire collection for this particular user.

```
<report type="stale_everything"/>
```

-->

```
<!-- ===== instances and blocks -->
```

```
<!ELEMENT instances (block+) >  
<!ATTLIST instances extent (partial|complete) "complete">
```

```
<!ELEMENT block (facet* item-def*) >  
<!ATTLIST block action (insert|update|delete|define) "insert"  
    fgcolor CDATA #IMPLIED  
    bgcolor CDATA #IMPLIED  
    hicolor CDATA #IMPLIED  
    uncolor CDATA #IMPLIED>
```

<!-- This section describes the instance information on items. A server can

ask a client to insert, update, delete, or define items within the collection.

To do that the server issues an instances statement, within which are one

or more blocks. Each block in turn contains zero or more item-defs, and

its action attribute specifies the action to perform on all items within

the block.

When the instances' extent is specified to be "complete", all the items

of the collection are listed in the block; they can be defined only

within a block that has an "insert" action. The client can assume that any missing item has been deleted.

Blocks are syntactical shorthand, a way of grouping items that have

common attribute values. A block's attribute values are applied to all

items within it, except for those attributes that are overridden by

individual items.

The same rule applies to a block's facets: whatever facets are defined

within a block are shared by all of the block's items, though individual

items may override a block's facet by defining a facet of the same kind.

Block facets are especially useful for defining logos to be shared by many different items.

It is an unchecked runtime error if two items with the same id are

simultaneously in two blocks with the same action.

When applying a block attribute value to an item is problematic, the

client will take appropriate actions (as defined below). It will also mark

the items in question in problem reports that are passed back to the server

in the next rendezvous.

-->

```
<!-- ===== block action
n="insert" -->
```

<!--

When the server wants to insert a new media item in client's cache, it will issue a block statement with its action set to "insert".

```

<block
  action="insert">
  <item
    id=[a item id]
    pos=...          ...position within the collection.

    ...
    content=...
    name=...
    details=...
    target=...
    rating="-1"
    type="mime/jpeg"
    info="100x100 pix, 25k"  ...interpreted by the screenplay..

    />
</block>

```

If the same item already exists in the collection, then the client:

- updates the values as per insert record
- marks the item for report with type "duplicate_item_insert".

If an item exists in this position then the client:

- inserts the current record at the end of the collection
- marks the item for report in the next rendezvous with type "index_collision"

<-->

```
<!-- ===== block action
n="update" -->
```

<!--

The update element is useful for changing values associated with an image. In particular, you can change the content URL for a parti-

cular
image (to deal with re-organizations of external sites where images
may live.) This is done by overriding the "content" element of the update
record.

```
<block  
  action="update">  
<item-def id=[a item id]  
  pos=  
  content=...  
  name=...  
  details=...  
  page=....  
  thumb=...  
  rating="-1"  
  type="mime/jpeg"  
  info="100x100 pix, 25k"  
  relmod="25"  
/>  
</block>
```

If the item referred to by "id" doesn't exist,
client must:

- create the item
- update its fields as specified in the transmission
- mark item for report of type "unknown_item_update"

If an item with a differnt id is located in the
same position as specified by the update:

- the position is set to the last item in the
collection
- the item is marked for report of type
"index_collision"

If the update conflicts with one made by the client (for example
a rating change):

- use the relmod + local time of rendezvous when we received this
update to determine which took place later.
- mark item for report of type "update_conflict"

-->

```
<!-- ===== block action="delete" -->
```

<!--

By sending a block with action="delete", the server requests the client

to delete a media item from the collection.

```
<block
  action="delete">
  <item-def id="..."/>
</block>
```

If the item doesn't exist, client marks it for report of type "unknown_item_delete".

-->

```
<!-- ===== block action="define" -->
```

<!--

A define action is just like an insert action, with the following exceptions:

- it can only be used in the "provide" mode
- it can only contain media items from URLs that are descendants of the parent URL of the PXP file. (This restriction makes it possible for people to create their own collections by creating a file or script on their own servers. However, these collections are static and cannot refer to other's contents.

-->

```
<!-- ===== url handling -->
```

```
<!--
  URLs passed onto the client may be relative to the Pixxa server,
  e.g., /client/customize?xyz=abc. When following this type of link
  (for example, to start a browser) the client must append the protocol and the hostname of the server (e.g.,
  http://dev.pixxa.com) which it is currently corresponding.
  Also, the query pxp_email=[user's email] is appended to the
  server-relative URLs, so that /client/customize?xyz=abc maps to
  o
  http://dev.pixxa.com/client/customize?xyz=abc&pxp_email=farshad@cmass.com

-->

<!-- ===== text handling -->

<!--
  Because of a limitation of XML, all text sent down will be
  URL-encoded.

  - & for ampersand (&)
  - &quot; for double quotes (")

  These markups should be unescaped before text handed by the server
  is processed by the client.

  So, if the original text is 1 & 2,
  the escaped text becomes 1 & 2
  and the client should eventually map this back to the original
  form.

-->

<!-- ===== screenplay parameters -->

<!--
```

Screenplay parameters (specified as a var-def named "params" within a special element) is a list of key-value bindings.

The format for the screenplay parameters is the same as HTTP query

parameters. (Note that non-alphanumeric values may be URLencoded;

also, since XML does not allow literals to carry ampersands they are replaced by the XML directive for ampersand .)

The key "transition" can be bound to one of:

wiperight
wipeleft
wipedown
wipeup
centerouth
edgesinh
centeroutv
edgesinv
centeroutsquare
edgesinsquare
pushleft
pushright
pushdown
pushup
revealup
revealupr
revealr
revealdownr
revealdown
revealdownl
revealleft
revealupl
dxpixelsfast
dxboxyrect
dxboxysquare
dxpatterns
randomrows
randomcols
coverdown
coverdownl

coverdownr
coverleft
coverright
coverup
coverupl
coverupr
venetian
checkerboard
stripbottoml
stripbottomr
stripleftdown
stripleftup
striprightdown
striprightup
striptopl
striptopr
zoomopen
zoomclose
vertblinds
dxbitsfast
dxpixels
dxbits

Not all clients may implement these transitions.
Depending on the client, there may also be other
parameters for the screenplay, for example, the
duration of the transition.

-->

<!-- ===== v3 restrictions -->

<!--

A valid v3 implementation of the protocol may place the following
restrictions:

1. rating specifications range from -1..0..+1.
2. A media item id is the same as content URL, but neither the client
nor the server can assume this.

3. var-def's type may only be "text" and "password" and "inform"
4. var-def password responses need not be encrypted
5. Neither the client nor the server need to worry about server-side reports.

→